Chapter 4

Unix Tutorial

Users and applications interact with hardware through an operating system (OS). Unix is a very basic operating system in that it has just the essentials. Many operating systems, such as macOS and Linux, are "Unix-like" in that they are built on top of it. All these systems are then closely compatible at the Unix level—knowing Unix is a versatile skill.

The user interface for Unix is a command-line *shell*, which we'll be using to do operations on the computer's *file system*. This may feel tedious at first, after all, you've managed files up until now without it. However, the advantage of Unix is in writing scripts that automate repetitive tasks. This enables you to work accurately with arbitrarily many files of whatever size.

Unix Shell

If you're using macOS or Linux, search your computer for the "Terminal."[1] This an application that runs *shells*, which here are command-line interfaces. The default will generally be a Unix shell called the "Bash." The current line of the Bash will prompt you with the following,

```
[Computer's name]:~ [Working directory][Username]$
```

Commands are typed after the dollar sign ($). We will optionally suppress the information in the prompt by typing `PS1='$ '` and pressing the "enter" key.

```
[Computer's name]:~ [Working directory][Username]$ PS1='$ '
$
```

[1] For other operating systems ask Google—your best friend when programming. Windows is not Unix-like, but you can download "Git" which will emulate the Terminal environment.

The current line is now prompted by '$ '. For future reference, `ctrl+C` stops any commands that are still running, and the Bash session can be ended with `ctrl+D`. (These are useful if you're stuck in a loop for instance.)

File system

Everything on your computer is stored in directories (aka folders). They form a tree hierarchy; each directory has a unique parent that contains it—with the exception of the *root*, which has no parent. The Bash has a *current working directory* where commands are executed from. The following three commands will allow you to navigate through your files: "pwd," "ls," and "cd." These are some of the most frequent commands you'll use. "Print working directory," `pwd`, gives the full path of the current working directory.

```
$ pwd
/Users/JohnCleese
```

This says the current working directory is "JohnCleese" inside of the "User" directory, which is itself inside of the root directory. (The current user is generally the default directory.) Use this command frequently to double check that you are in the directory you mean to be in. "List" files, `ls`, prints the files in the current working directory. For example, the user's directory will typically look like,

```
$ ls
Applications                    Library
Desktop                         Movies
Documents                       Pictures
Downloads                       Public
```

"Change directory," `cd`, navigates the file system. Suppose we want to be in the "applications" directory, we need only type,

```
$ cd applications
```

To confirm that this worked, run "pwd."

```
$ pwd
/Users/JohnCleese/applications
```

The computer knew where to find "applications" because it was in the current working directory. Otherwise, the path of the new working directory must be inputted. For instance, the following would go to the "desktop" directory.

```
$ cd /Users/JohnCleese/desktop
$ pwd
/Users/JohnCleese/desktop
$ ls
Fawlty_Towers.pdf          Monty Python          Text.txt
```

Unix has shortcuts for various directories. Along with being easier to read, these shortcuts can make code robust to where it is being executed.

- `.` is the "current working directory." (/Users/JohnCleese/desktop)
- `..` is the "parent directory." (/User/JohnCleese)
- `-` is the "previous working directory." (/Users/JohnCleese/applications)
- `~` is the default "user's directory." (/Users/JohnCleese)

For example, we could write `cd -` to go back to the "applications" directory. Notice that no input to the "cd" command takes us back to the default user's directory.

```
$ cd
$ pwd
/Users/JohnCleese
```

Whitespace

The Bash uses whitespace to separate multiple arguments. If a name involves spaces, then it must be put in quotes on the command line. Otherwise, the name will be split into distinct arguments, and the Bash will give a corresponding error.

```
$ cd ~/desktop/Monty Python
-bash: cd: /Users/JohnCleese/desktop/Monty: No such file or directory
$ cd ~/desktop/'Monty Python'
$ pwd
/Users/JohnCleese/desktop/Monty Python
```

The best way to avoid errors caused by whitespace is simply not to use it within names, e.g. use '_' instead.

Flags

Commands often have various options for how they are used. These options can be selected by adding *flags* between the command and any input. For instance, `ls -F` appends a "/" to directories in its printout.

```
$ cd ~/desktop
$ ls -F
Fawlty_Towers.pdf         Monty Python/         Text.txt
```

There are many files on your computer that are suppressed from view. `ls -a` will show all files in your current directory. (Try this command in your default directory and you might be surprised by the number of such files.) Multiple flags can be concatenated, for instance `ls -a -F`. Typing `man` before a command takes you to its manual page, e.g. `man ls`. Among much else, this page will explain all optional flags. Press "q" to exit the manual back to the command line. (Alternative commands to `man` are `info` and `help`.)

Make, move, copy, and remove

"Make directory," `mkdir`, creates a new directory.

```
$ cd ~/desktop/'Monty Python'
$ mkdir Silly_Walks
$ ls -F
Silly_Walks/
```

The `touch` command "touches" a file; if the file did not exist, then it creates it. (Otherwise, it simply changes the "modification" and "access" times.)

```
$ touch Parrot.sh
$ ls -F
Parrot.sh       Silly_Walks/
```

In this example we created the blank shell script "Parrot.sh." Any kind of blank text file can be created this way. "Move," `mv`, can change the directory of a file as well as its name. The first input specifies a current file, and the second input specifies where to move it. For example,

```
$ mv Parrot.sh ../Ex_Parrot.sh
```

This moved the file "Parrot.sh" to the parent directory and changed the name to "Ex_Parrot.sh." Be careful not to accidentally overwrite files; if there had already been such a file, then it

would have been overwritten. The `-n` flag prevents overwriting (see `man mv` for more information). "Copy" files, `cp`, works the same way as `mv` except that the original file is not deleted. For example,

```
$ cp ../Ex_Parrot.sh Copy_of_Ex_Parrot.sh
```

This created a copy of "Ex_Parrot.sh" called "Copy_of_Ex_Parrot.sh" in the current directory. "Remove," `rm`, permanently[2] deletes a file.

```
$ rm Copy_of_Ex_Parrot.sh
$ ls
Silly_Walks
```

Note that the flag "-d" must be used to delete a directory. If the directory is not empty, then the flag "-r" must be used to delete all files within it. (Be especially careful when using this flag.)

Output

The `echo` command outputs text.

```
$ echo Hello world!
Hello world!
```

The text can be directed to a file using `>` or `>>`. If the file doesn't exist, then it will be created.

```
$ echo [some text 1] > example.txt
$ echo [some text 2] >> example.txt
```

The `>` overwrites any such file, whereas `>>` appends the text to end of the file. (The opposite direction `<` can be used to input files to commands.) For those curious, the Bash provides its own inline plain text editor called `nano`. (This won't be used for automation, so there's not much reason to use this instead of the text editor application on your computer.)

Arithmetic

Arithmetic can be done within double parentheses.

```
$ echo $((8+2))
10
```

[2] It is deleted and does not go into a trash bin, so be careful when using this command.

```
$ echo $((7*3))
21
$ echo $((23/4))
5
```

Often an expression needs to be preceded by a dollar sign ($). Otherwise, "echo" would simply print the expression ((8+2)) instead of the result of 10. Alternatively, evaluate "expression," `expr`, can be used.

```
$ expr 7 - 12
-5
$ expr 13 \* 5
65
```

Loops

The following `for` loop prints out the numbers 1 to 5 on separate lines.

```
$ for i in 1 2 3 4 5
> do
> echo $i
> done
```

Instead of listing out the values of i, the for loop can be written as in C and Java.

```
$ for ((i=1; i<=5; i++))
> do
> echo $i
> done
```

Semicolons can be used to put multiple commands on the same line. The following for loop is equivalent to the previous.

```
$ for ((i=1; i<=5; i++)); do echo $i; done
```

A `while` loop is very similar, but instead evaluates a logical statement.

```
$ while logical statement; do ... ; done
```

Logic and if statements

"If statements" have the following structure.

```
$ if logical statement 1; then
>       ...
> elif logical statement 2; then
>       ...
> else
>       ...
> fi
```

The following are logical operators in mathematical expressions.

- `<` is "less than," and `<=` is less than or equal to.
- `>` is "greater than," and `>=` is greater than or equal to.
- `==` is "equal to," and `!=` is "not equal to."

```
$ if ((1 <= 2)); then echo true; else echo false; fi
true
$ if ((5 == 3)); then echo true; else echo false; fi
false
```

Strings can be compared like numbers, but they must be enclosed in brackets, e.g.

```
$ if [ 'ABC' != 'XYZ' ]; then echo true; else false; fi
true
```

Logical statements can be concatenated with "and" `&&`, and "or" `||`.

Prime numbers example

In the following toy project, we are going to make a shell script that computes all of the prime numbers up to an arbitrary N. First, we'll make a new directory for the project. Start in whatever directory you like, e.g. you desktop.

```
$ mkdir Prime_Calculator
$ cd Prime_Calculator
$ touch Times_Table.sh
$ touch Calculator.sh
```

We'll use the "Times_Table" script to output all composite numbers less than N. Then, the "Calculator" script will find all of the primes.

Shell scripts

Shell scripts are plain text files with the extension `.sh`. Unix commands can be written in shell scripts the same way they would be written in the Bash. Consider the following "Times_Table" script. (Copy and paste into the file.)

```
### This is a helper script for computing prime numbers.
# For each number i less than sqrt(N), this script computes its multiples.
# These multiples are output as individual text files in the Table directory.
# The only input is N.
rm -r Table   # Removes the old multiplication table.
mkdir Table   # Creates a new multiplication table directory.
# For all numbers i less than N, compute the multiples of i.
for ((i=2; i<=$1; i++))
do
  # Consider all j-th multiples of i (between i and N/i).
  for ((j=i; j<=($1/i); j++))
  do # Output i*j as a multiple of i.
    echo $((i*j)) >> "Table/Multiples_of_$i.txt"
  done
done
```

Inputs are referenced in order as \$1, \$2, ... within the script. Our script only takes in one input, N, which is denoted by \$1. The `bash` command runs shell scripts.

```
$ bash Times_Table.sh 100
```

This ran the "Times_Table" script with $N = 100$.

Viewing files

Let's inspect the output in the "Table" directory.

```
$ cd Table
$ ls
Multiples_of_10.txt    Multiples_of_4.txt    Multiples_of_7.txt
Multiples_of_2.txt     Multiples_of_5.txt    Multiples_of_8.txt
Multiples_of_3.txt     Multiples_of_6.txt    Multiples_of_9.txt
```

There are many commands for viewing files in the Bash. `cat` prints out the contents of a file. `head` prints out the first ten lines unless you specify a different number. Whereas, `tail` prints out the last lines of a file.

```
$ head -3 Multiples_of_2.txt
4
6
8
```

Wildcards

"Word count," `wc`, computes the total number of words in a file. Supposing we were interested in the total number of each such multiple, the following would compute it.

```
$ wc -l *.txt
```

The asterisk (*) is a *wildcard* for any valid expression. In this example, the word count command took every ".txt" file in the directory as input. Now, a question mark (?) is a wildcard for any single character.

```
$ wc -l Multiples_of_?.txt
```

This computes the word count for the multiples of any single digit.

Pipes

"Sort numerically," `sort -n`, takes a list of numbers and sorts it in ascending order.

```
$ sort -n *.txt
```

This took all of the numbers in all of the text files and sorted them. Note that there will be plenty of duplicates for numbers with multiple factors. "Unique," `uniq`, eliminates any adjacent duplicates within a list. Apply this command after sorting will eliminate any duplicates. This is accomplished with *pipes* (|).

```
$ sort -n *.txt | uniq
```

The lefthand side of the pipe (|) is computed first and then inputted into the command on the righthand side. In this example, the output is all of the composite numbers less than $N = 100$.

Calculator

The final step in computing the prime numbers is to take the complement of the composite numbers. After a couple quick Google searches, you would find that,

```
$ seq 2 100
```

lists all the integers from 2 to 100. And, `comm -23` takes the set minus of two lists. Hence, the following gives all the primes under 100,

```
$ comm -23 <(seq 2 100) <(sort -n *.txt | uniq)
```

Note that the $<(\ldots)$ is being used to group as a single input. To automate this calculation, we simply have to write a script that executes this last command after calling "Times_Table.sh." Our "Calculator.sh" code is:

```
bash Times_Table.sh $1
echo $(comm -23 <(seq 2 $1) <(sort -n Table/*.txt | uniq)) > Primes.txt
```

Woohoo! Now you can compute all of the prime numbers less than N by calling,

```
$ bash Calculator.sh N
```

in your "Prime_Calculator" directory.